CS 6363.003.21S Lecture 20—April 15, 2021

Main topics are #max_flow-min_cut_applications.

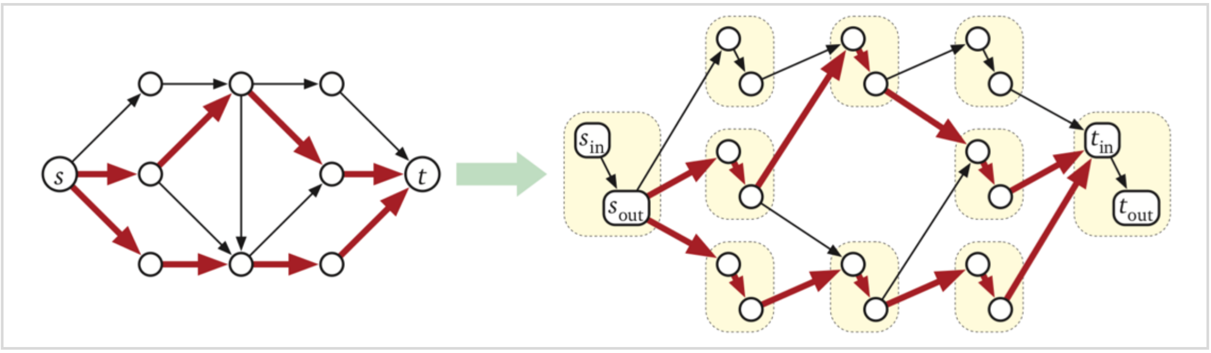
Edge-Disjoint Paths

- It turns out the maximum flow and minimum cut problems are useful subroutines for several other problems. Let's start with a straightforward one:
- First, suppose we have a directed graph G = (V, E) and two vertices s and t.
- The *edge-disjoint paths* problem asks for a maximum size set of paths from s to t where every edge appears in at most one path.
- This problem is really easy to solve via a reduction to maximum flows.
- Give each edge capacity 1.
- Any collection of k edge-disjoint paths can be made into a flow of value k by assigning 1 to each edge along the paths.
- So the value of a maximum flow f^* is at least the maximum number of edge disjoint paths.
- But now, suppose we compute a maximum flow f^*.
- Because the capacities are integers, we may assume flow f^* assigns an integer to every edge. The only options are 0 or 1.
- We can actually extract | f^* | paths, which we just learned is the most we can get.
- Let S be the set of edges with a 1. We can return a path from s to t in S, remove the edges of that path from S, and then recursively find | f^* | - 1 paths from what remains for | f^* | paths total.
- In short, finding the number of edge-disjoint paths is as simple as computing a maximum flow and returning its value.
- We could run Orlin's algorithm in O(VE) time, but it's actually overkill assuming the graph is simple. Instead, notice that the cut ({s}, V \ {s}) has capacity at most V - 1. So | f^* | is at most V - 1. Standard Ford-Fulkerson with arbitrary augmenting paths will take O(VE) time as well.

Vertex Capacities and Vertex-Disjoint Paths

- Let's go back to computing flows, but suppose now we want to limit amount of flow going through *vertices* instead of just edges. i.e., we have capacities c : V ➡ R_≥0 and a flow is feasible if sum_{u ➡ v} f(u ➡ v) ≤ c(v) for all v ≠ s, t.
- There's a simple reduction we can perform in O(E) time to the standard problem with only edge capacities.
- Replace every vertex v with two vertices v_in and v_out connected by an edge v_in ➡ v_out with capacity c(v). Replace every directed edge u ➡ v with an edge u_out ➡ v_in,

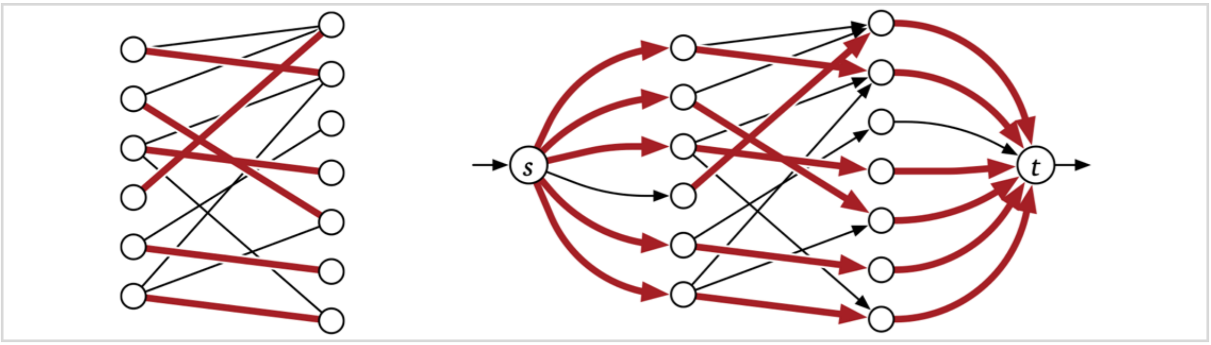
keeping the same capacity. Now flow going into any v must equal flow going through v_in ➜ v_out so we're enforcing vertex capacities! Any feasible flow in the original graph is feasible in the new one (after filling in the v_in ➜ v_out edges) and vice versa, so it suffices to compute maximum flows in the new graph.



- In particular, we can now solve *vertex-disjoint paths* too. Assign a capacity of 1 to every vertex and run the above algorithm.

Bipartite Matching

- Alright, so flows can help you find paths that don't use edges or vertices too many times. Maybe that's not so surprising.
- But now, suppose we have an undirected bipartite graph G = (L cup R, E) where L and R are disjoint and every edge is incident to a vertex in L and a vertex in R. A *matching* in G is a subgraph in which every vertex has degree at most one. In other words, we pair up some vertices between L and R, but no vertex appears in more than one pair.
- We want to find a matching with the maximum number of edges. Maybe L is a set of medical students and R is a set of slots in residency programs. There are edges between compatible assignments, and we want as many assignments as possible.
- We can use flows after modifying the graph a bit. We create a directed graph G' by
  - orienting edges from L to R
  - adding new vertices s and t
  - adding edges from s to every vertex in L
  - adding edges from every vertex in R to t
  - giving every edge in G' a capacity of 1



- Any matching M can be turned into a flow f_M in G'. For each edge uw in M (u in L and w in

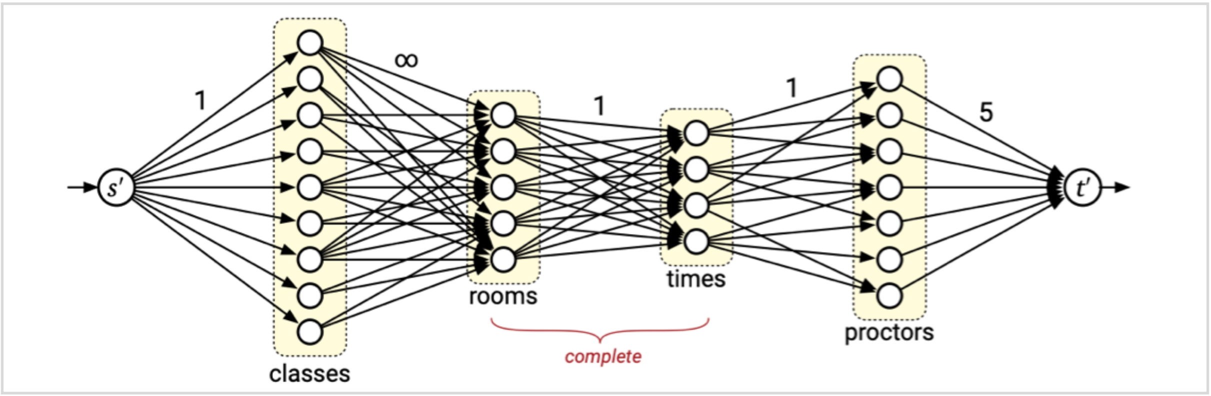
R), push one unit of flow along s ➡ u ➡ w ➡ t. We have | f_M | = | M |.

- But now say we compute a maximum flow f^*. We now know |f^ *| ≥ |M^ *| where M^ * is the maximum cardinality matching. But can we fine a matching that large?
- The capacities are integers, so every edge is assigned an integral value of flow, either 0 or 1.
- Every vertex in L has one incoming edge, so at most 1 unit of incoming flow. So there is at most one outgoing edge with 1 unit of flow. Similarly, every member of R has at most 1 incoming edge with one unit of flow.
- So the edges from L to R with 1 unit form a matching. And they carry all | f^* | units of flow, so the matching has size |f^ *|.
- Again, Orlin is overkill here, because there are at most V / 2 edges in any matching. Just run Ford-Fulkerson with arbitrary augmenting paths in O(VE) time.

Exam Scheduling

- I gave an example where the matching can be thought of as an assignment. We can extend these ideas to assign multiple kinds of things at once.
- Suppose we've been asked to schedule the final exams for next Fall. We'll assume the exams are being given on campus. There are
  - n classes,
  - r rooms,
  - t time slots, and
  - p proctors to oversee the exams.
- At most one class's final exam can be scheduled in each room during each time slot, and classes cannot be split between multiple rooms and time slots.
- Each proctor can
  - oversee one exam at a time,
  - is available for only certain time slots,
  - can oversee at most 5 exams total.
- And of course, we have to worry about whether students can even fit in their assigned rooms!
- Here's all the input:
  - Integer array E[1 .. n] where E[i] is the number of students enrolled in class i.
  - Integer array S[1 .. r] where S[j] is the number of seats in rooms j. So class i's final can be held in room j if and only if E[i] ≤ S[j].
  - Boolean array A[1 .. t, 1 .. p] where A[k, ell] = True if and only if proctor ell is available during the kth time slot.
- Wow, that's a lot to take in!
- But there's a pretty slick reduction to maximum flow, just like before.

- We'll construct a graph G with *six* types of vertices:
  - source vertex s'.
  - a vertex c_i for each class i
  - a vertex r_j for each room j
  - a vertex t_k for each time slot
  - a vertex p_ell for each proctor, and
  - a target vertex t'
- There are five types of edges:
  - an edge s ➜ c_i with capacity 1 for each class (one final exam per class),
  - an infinite capacity edge c_i ➜ r_j for each class i and room j such that E[i] ≤ S[j] (class i can fit in room j if there are more students than seats)
  - an edge r_j ➜ t_k with capacity 1 for each room j and time slot k (at most one exam can be held in room j at time k)
  - an edge t_k ➜ p_ell with capacity 1 for time slot k and proctor ell such that A[ell, k] = True (a proctor can oversee one exam at a time, and only when they are available)
  - an edge p_ell ➜ t' with capacity 5 for each proctor ell (each proctor can oversee at most 5 exams)
- So G has n + r + t + p + 2 vertices and O(nr + rt + tp) edges.



- As you might guess, we're going to compute a maximum (s',t')-flow.
- If we're given an assignment of class i, room j, time k, and proctor ell, we can map it to a path s' ➜ c_i ➜ r_j ➜ t_k ➜ p_ell ➜ t'.
- So, we can take a correct assignment for all the classes, and it will map to a flow of value n. There's n paths out of s. One per class. One per room-time slot pair. One per time slot-proctor pair. And at most 5 per proctor. So if we assign to each edge the number of times it appears in one of these paths, we get a feasible (s',t')-flow of value n.
- So if we compute a maximum (s',t')-flow f*, it will have value at least n if there is any way to assign all the exams.
- In fact, f^* will have value exactly n, because there's that cut ({s}, V \ {s}).
- And like before, we may assume f^* has integral values. So we'll peel off one path, reduce the flow on each edge of the path by 1, and recurse to get all the assignments.
- Again, arbitrary path Ford-Fulkerson works just fine here, so the whole thing will take O(VE)

$= O((n + r + t + p)(nr + rt + tp))$ time.

- You can solve many kinds of "assignment" style problems in this way. Make one set of vertices per type of object. Add edges between objects that can be assigned to one another. Add some capacities if some pair or some object can be involved in a limited number of assignments.
- The lecture notes provide a bit more of a framework, but this is another one of those things you kind of get a feel for.
- And with that, we're done discussing graph algorithms.
- And in a way, we're done discussing efficient algorithm design.
- But the semester isn't over yet! Next Tuesday, we finally learn about that NP-hardness thing that keeps coming up. (Then review, then Midterm 2, then more NP-hardness to close the semester.)